答疑3 | 第13~18讲课后思考题答案及常见问题答疑

2021-10-30 蒋德钧

《Redis源码剖析与实战》 课程介绍 >

讲述：蒋德钧
时长 15:36 大小 14.29M

你好，我是蒋德钧。

今天这节课，我们继续来解答第 13 讲到第 18 讲的课后思考题。这些思考题除了涉及 Redis 自身的开发与实现机制外，还包含了多线程模型使用、系统调用优化等通用的开发知识，希望你能掌握这些扩展的通用知识，并把它们用在自己的开发工作中。

领资料

第 13 讲

问题：Redis 多 IO 线程机制使用 startThreadedIO 函数和 stopThreadedIO 函数，置 IO 线程激活标识 io_threads_active 为 1 和为 0。此处，这两个函数还会对线程互斥锁数组进行解锁和加锁操作，如下所示。那么，你知道为什么这两个函数要执行解锁和加锁操作吗？

复制代码

```
void startThreadedIO(void) {
    ...
    for (int j = 1; j < server.io_threads_num; j++)
        pthread_mutex_unlock(&io_threads_mutex[j]);  //给互斥锁数组中每个线程对应的
    server.io_threads_active = 1;
}

void stopThreadedIO(void) {
    ...
    for (int j = 1; j < server.io_threads_num; j++)
        pthread_mutex_lock(&io_threads_mutex[j]);  //给互斥锁数组中每个线程对应的互
    server.io_threads_active = 0;
}
```

我设计这道题的目的，主要是希望你可以了解多线程运行时，如何通过互斥锁来控制线程运行状态的变化。这里我们就来看下线程在运行过程中，是如何使用互斥锁的。通过了解这个过程，你就能知道题目中提到的解锁和加锁操作的目的了。

首先，在初始化和启动多 IO 线程的 **initThreadedIO 函数**中，主线程会先获取每个 IO 线程对应的互斥锁。然后，主线程会创建 IO 线程。当每个 IO 线程启动后，就会运行 IOThreadMain 函数，如下所示：

复制代码

```
void initThreadedIO(void) {
   …
    for (int i = 0; i < server.io_threads_num; i++) {
  …
  pthread_mutex_init(&io_threads_mutex[i],NULL);
   io_threads_pending[i] = 0;
   pthread_mutex_lock(&io_threads_mutex[i]); //主线程获取每个IO线程的互斥锁
   if (pthread_create(&tid,NULL,IOThreadMain,(void*)(long)i) != 0) {…} //启动IO
  …} …}
```

而 IOThreadMain 函数会一直执行一个 **while(1)的循环流程。在这个流程中，线程又会先执行一个 100 万次的循环，而在这个循环中，线程会一直检查有没有待处理的任务，这些任务的数量是用io_threads_pending 数组**保存的。

在这个 100 万次的循环中，一旦线程检查到有待处理任务，也就是 io_threads_pending 数组中和当前线程对应的元素值不为 0，那么线程就会跳出这个循环，并根据任务类型进

行实际的处理。

下面的代码展示了这部分的逻辑，你可以看下。

```
复制代码
1  void *IOThreadMain(void *myid) {
2   …
3    while(1) {
4          //循环100万次，每次检查有没有待处理的任务
5           for (int j = 0; j < 1000000; j++) {
6               if (io_threads_pending[id] != 0) break;  //如果有任务就跳出循环
7           }
8           … //从io_threads_lis中取出待处理任务，根据任务类型，调用相应函数进行处理
9           }
10 …}
```

而如果线程执行了 100 万次循环后，仍然没有任务处理。那么，它就会**调用 pthread_mutex_lock 函数**去获取它对应的互斥锁。但是，就像我刚才给你介绍的，在 initThreadedIO 函数中，主线程已经获得了 IO 线程的互斥锁了。所以，在 IOThreadedMain 函数中，线程会因为无法获得互斥锁，而进入等待状态。此时，线程不会消耗 CPU。

与此同时，主线程在进入事件驱动框架的循环前，会**调用 beforeSleep 函数**，在这个函数中，主线程会进一步调用 handleClientsWithPendingWritesUsingThreads 函数，来给 IO 线程分配待写客户端。

那么，在 handleClientsWithPendingWritesUsingThreads 函数中，如果主线程发现 IO 线程没有被激活的话，它就会**调用 startThreadedIO 函数**。

好了，到这里，startThreadedIO 函数就开始执行了。这个函数中会依次调用 pthread_mutex_unlock 函数，给每个线程对应的锁进行解锁操作。这里，你需要注意的是，startThreadedIO 是在主线程中执行的，而每个 IO 线程的互斥锁也是在 IO 线程初始化时，由主线程获取的。

所以，主线程可以调用 pthread_mutex_unlock 函数来释放每个线程的互斥锁。

一旦主线程释放了线程的互斥锁，那么 IO 线程执行的 IOThreadedMain 函数，就能获得对应的互斥锁。紧接着，IOThreadedMain 函数又会释放释放互斥锁，并继续执行 while(1)，如下所示：

复制代码

```
void *IOThreadMain(void *myid) {
 …
  while(1) {
     …
     if (io_threads_pending[id] == 0) {
            pthread_mutex_lock(&io_threads_mutex[id]);  //获得互斥锁
            pthread_mutex_unlock(&io_threads_mutex[id]); //释放互斥锁
            continue;
     }
   …} …}
```

那么，这里就是解答第 13 讲课后思考题的关键所在了。

在 IO 线程释放了互斥锁后，主线程可能正好在执行 handleClientsWithPendingWritesUsingThreads 函数，这个函数中除了刚才介绍的，会根据 IO 线程是否激活来启动 IO 线程之外，它也会**调用 stopThreadedIOIfNeeded 函数，来判断是否需要暂停 IO 线程**。

stopThreadedIOIfNeeded 函数一旦发现待处理任务数，不足 IO 线程数的 2 倍，它就会调用 stopThreadedIO 函数来暂停 IO 线程。

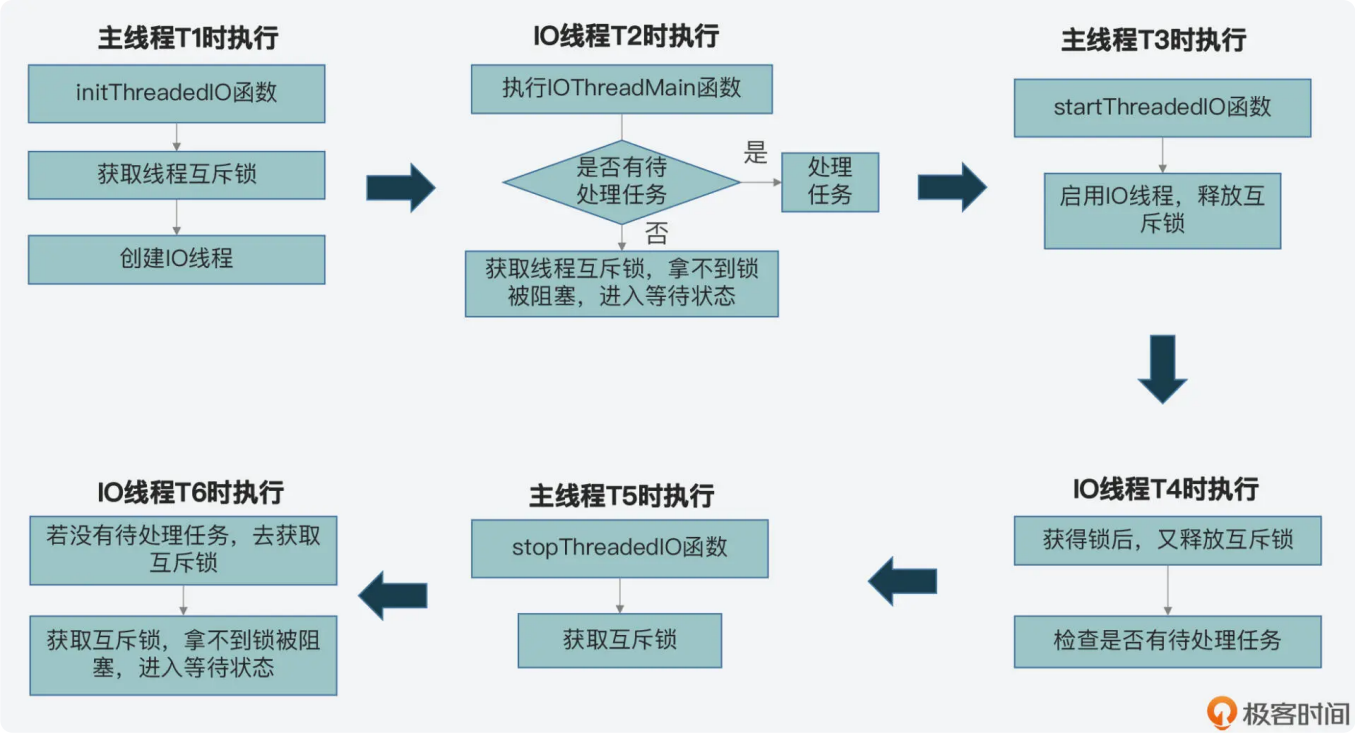
而暂停 IO 线程的办法，就是让主线程获得线程的互斥锁。所以，stopThreadedIO 函数就会依次调用 pthread_mutex_lock 函数，来获取每个 IO 线程对应的互斥锁。刚才我们介绍的 IOThreadedMain 函数在获得互斥锁后，紧接着就释放互斥锁，其实就是希望主线程执行的 stopThreadedIO 函数，能在 IO 线程释放锁后的这个时机中，获得线程的互斥锁。

这样一来，因为 IO 线程执行 IOThreadedMain 函数时，会一直运行 while(1) 循环，并且一旦判断当前待处理任务为 0 时，它会去获取互斥锁。而此时，如果主线程已经拿到锁了，那么 IO 线程就只能进入等待状态了，这就相当于暂停了 IO 线程。

这里，你还需要注意的一点是，**stopThreadedIO 函数还会把表示当前 IO 线程激活的标记 io_threads_active 设为 0**，这样一来，主线程的 handleClientsWithPendingWritesUsingThreads 函数在执行时，又会根据这个标记来再次调用 startThreadedIO 启用 IO 线程。而就像刚才我们提到的，startThreadedIO 函数会释放主线程拿的锁，让 IO 线程从等待状态进入运行状态。

关于这道题，不少同学都提到了，题目中所说的加解锁操作是为了控制 IO 线程的启停，而且像是 @土豆种南城同学，还特别强调了 IOThreadedMain 函数中执行的 100 万次循环的作用。

因为这个题目涉及的锁操作在好几个函数间轮流执行，所以，我刚才也是把这个过程的整体流程给你做了解释。下面我也画了一张图，展示了主线程通过加解锁控制 IO 线程启停的基本过程，你可以再整体回顾下。



第 14 讲

问题：如果我们将命令处理过程中的命令执行也交给多 IO 线程执行，你觉得除了对原子性有影响，会有什么好处或其他不足的影响吗？

这道题主要是希望你能对多线程执行模型的优势和不足，有进一步的思考。

其实，使用多 IO 线程执行命令的好处很直接，就是可以充分利用 CPU 的多核资源，让每个核上的 IO 线程并行处理命令，从而提升整体的吞吐率。

但是，这里你要注意的是，如果多个命令执行时要对同一个数据结构进行写操作，那么，此时也就是多个线程要并发写某个数据结构。为了保证操作正确性，我们就需要使用**互斥方法**，比如加锁，来提供并发控制。

这实际上是使用多 IO 线程时的不足，它会带来两个影响：一个是基于加锁等互斥操作的并发控制，会降低系统整体性能；二个是多线程并发控制的开发与调试比较难，会增加开发者的负担。

第 15 讲

问题：Redis 源码中提供了 getLRUClock 函数来计算全局 LRU 时钟值，同时键值对的 LRU 时钟值是通过 LRU_CLOCK 函数来获取的，以下代码也展示了 LRU_CLOCK 函数的执行逻辑，这个函数包括了两个分支，一个分支是直接从全局变量 server 的 lruclock 中获取全局时钟值，另一个是调用 getLRUClock 函数获取全局时钟值。

那么你知道，为什么键值对的 LRU 时钟值，不直接通过调用 getLRUClock 函数来获取呢？

复制代码

```
unsigned int LRU_CLOCK(void) {
    unsigned int lruclock;
    if (1000/server.hz <= LRU_CLOCK_RESOLUTION) {
        atomicGet(server.lruclock,lruclock);
    } else {
        lruclock = getLRUClock();
    }
    return lruclock;
}
```

这道题有不少同学都给出了正确答案，比如 @可怜大灰狼、@Kaito、@曾轼麟等等。这里我来总结下。

其实，调用 getLRUClock 函数获取全局时钟值，它最终会调用 **gettimeofday** 这个系统调用来获取时间。而系统调用会触发用户态和内核态的切换，会带来微秒级别的开销。

而对于 Redis 来说，它的吞吐率是每秒几万 QPS，所以频繁地执行系统调用，这里面带来的微秒级开销有些大。所以，**Redis 只是以固定频率调用 getLRUClock 函数**，使用系统调用获取全局时钟值，然后将该时钟值赋值给全局变量 server.lruclock。当要获取时钟时，直接从全局变量中获取就行，节省了系统调用的开销。

刚才介绍的这种实现方法，在系统的性能优化过程中是有不错的参考价值的，你可以重点掌握下。

第 16 讲

问题：LFU 算法在初始化键值对的访问次数时，会将访问次数设置为 LFU_INIT_VAL，它的默认值是 5 次。那么，你能结合这节课介绍的代码，说说如果 LFU_INIT_VAL 设置为 1，会发生什么情况吗？

这道题目主要是希望你能理解 LFU 算法实现时，对键值对访问次数的增加和衰减操作。**LFU_INIT_VAL 会在 LFULogIncr 函数中使用**，如下所示：

复制代码

```
uint8_t LFULogIncr(uint8_t counter) {
…
double r = (double)rand()/RAND_MAX;
    double baseval = counter - LFU_INIT_VAL;
    if (baseval < 0) baseval = 0;
    double p = 1.0/(baseval*server.lfu_log_factor+1);
  if (r < p) counter++;
  …}
```

从代码中可以看到，如果 LFU_INIT_VAL 比较小，那么 baseval 值会比较大，这就导致 p 值比较小，那么 counter++ 操作的机会概率就会变小，这也就是说，键值对访问次数 counter 不容易增加。

而另一方面，**LFU 算法在执行时，会调用 LFUDecrAndReturn 函数**，对键值对访问次数 counter 进行衰减操作。counter 值越小，就越容易被衰减后淘汰掉。所以，如果 LFU_INIT_VAL 值设置为 1，就容易导致刚刚写入缓存的键值对很快被淘汰掉。

因此，为了避免这个问题，LFU_INIT_VAL 值就要设置的大一些。

第 17 讲

问题：freeMemoryIfNeeded 函数在使用后台线程删除被淘汰数据的时候，你觉得在这个过程中，主线程仍然可以处理外部请求吗？

这道题像 @Kaito 等不少同学都给出了正确答案，我在这里总结下，也给你分享一下我的思考过程。

Redis 主线程在执行 freeMemoryIfNeeded 函数时，这个函数确定了淘汰某个 key 之后，会先把这个 key 从全局哈希表中删除。然后，这个函数会在 dbAsyncDelete 函数中，**调用 lazyfreeGetFreeEffort 函数，评估释放内存的代价。**

这个代价的计算，主要考虑的是要释放的键值对是集合时，集合中的元素数量。如果要释放的元素过多，主线程就会在后台线程中执行释放内存操作。此时，主线程就可以继续正常处理客户端请求了。而且因为被淘汰的 key 已从全局哈希表中删除，所以客户端也查询不到这个 key 了，不影响客户端正常操作。

第 18 讲

问题：你能在 serverCron 函数中，查找到 rdbSaveBackground 函数一共会被调用执行几次么？这又分别对应了什么场景呢？

这道题，我们通过在 serverCron 函数中查找 **rdbSaveBackground 函数**，就可以知道它被调用执行了几次。@曾轼麟同学做了比较详细的查找，我整理了下他的答案，分享给你。

首先，在 serverCron 函数中，它会直接调用 rdbSaveBackground 两次。

第一次直接调用是在满足 RDB 生成的条件时，也就是修改的键值对数量和距离上次生成 RDB 的时间满足配置阈值时，serverCron 函数会调用 rdbSaveBackground 函数，创建子进程生成 RDB，如下所示：

复制代码

```
1 if (server.dirty >= sp->changes && server.unixtime-server.lastsave > sp->secon
2 (server.unixtime-server.lastbgsave_try> CONFIG_BGSAVE_RETRY_DELAY
3 || server.lastbgsave_status == C_OK)) {
4 …
```

```
5 rdbSaveBackground(server.rdb_filename,rsiptr);
6 …}
```

第二次直接调用是在客户端执行了 BGSAVE 命令后，Redis 设置了 rdb_bgsave_scheduled 等于 1，此时，serverCron 函数会检查这个变量值以及当前 RDB 子进程是否运行。

如果子进程没有运行的话，那么 serverCron 函数就调用 rdbSaveBackground 函数，生成 RDB，如下所示：

复制代码

```
1 if (!hasActiveChildProcess() && server.rdb_bgsave_scheduled &&
2 (server.unixtime-server.lastbgsave_try > CONFIG_BGSAVE_RETRY_DELAY ||
3          server.lastbgsave_status == C_OK)) {
4  …
5 if (rdbSaveBackground(server.rdb_filename,rsiptr) == C_OK)
6 …}
```

而除了刚才介绍的两次直接调用以外，在 serverCron 函数中，还会有两次对 rdbSaveBackground 的**间接调用**。

一次间接调用是通过 replicationCron -> startBgsaveForReplication -> rdbSaveBackground 这个调用关系，来间接调用 rdbSaveBackground 函数，为主从复制定时任务生成 RDB 文件。

另一次间接调用是通过 checkChildrenDone –> backgroundSaveDoneHandler -> backgroundSaveDoneHandlerDisk -> updateSlavesWaitingBgsave -> startBgsaveForReplication -> rdbSaveBackground 这个调用关系，来生成 RDB 文件的。而这个调用主要是考虑到在主从复制过程中，有些从节点在等待当前的 RDB 生成过程结束，因此在当前的 RDB 子进程结束后，这个调用为这些等待的从节点新调度启动一次 RDB 子进程。

小结

好了，今天这节课就到这里了。我来总结下。

在今天的课程上，我给你解答了第 13 讲到第 18 讲的课后思考题。在这其中，我觉得有两个内容是你需要重点掌握的。

一个是你要了解系统调用的开销。从绝对值上来看，系统调用开销并不高，但是对于像 Redis 这样追求高性能的系统来说，每一处值得优化的地方都要仔细考虑。避免额外的系统开销，就是高性能系统开发的一个重要设计指导原则。

另一个是多 IO 线程模型的使用。我们通过思考题，了解了 Redis 会通过线程互斥锁，来实现对线程运行和等待状态的控制，以及多线程的优点和不足。现在的服务器硬件都是多核 CPU，所以多线程模型也被广泛使用，用好多线程模型可以帮助我们实现系统性能的提升。

所以在最后，我也再给你关于多线程模型开发的三个小建议：

尽量减少共享数据结构的使用，比如采用 key partition 的方法，让每个线程负责一部分 key 的访问，这样可以减少并发访问的冲突。当然，如果要做范围查询，程序仍然需要访问多个线程负责的 key。

对共享数据结构进行优化，尽量采用原子操作来减少并发控制的开销。

将线程和 CPU 核绑定，减少线程在不同核上调度时带来的切换开销。

欢迎继续给我留言，分享你在学习课程时的思考。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 答疑2 | 第7~12讲课后思考题答案及常见问题答疑

下一篇 答疑4 | 第19~24讲课后思考题答案及常见问题答疑

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。